Strong Normalization for Church-Style System F

Arthur Freitas Ramos David Barros Hulak
Ruy J. G. B. de Queiroz

September 17, 2026

Contents

Abstract

This entry formalizes strong normalization for Church-style System F with explicit type abstraction and type application. Types and terms use de Bruijn indices; the reduction relation contains both term-β and type-β steps and is closed under all term contexts. The proof combines Girard-style reducibility candidates [5] for an untyped lambda-calculus erasure with an explicit type-syntax measure that reflects the erased normalization result back to the full calculus. The final theorem states that every well-typed System F term is strongly normalizing, with the so the usual closed-term result is an immediate instance.

1 Overview

System F extends the simply typed lambda calculus with universal types, polymorphic abstraction, and type instantiation. The formalization represents these constructs explicitly:

$$\text{TmTAbs } M \qquad \text{and} \qquad \text{TmTApp } M\ A.$$

Typing uses de Bruijn indices for both term and type binders. The relation $\longrightarrow_\beta$ includes ordinary term-β, type-β, and compatible-context rules.

The semantic part interprets each type as a reducibility candidate in the untyped lambda calculus. Universal types quantify over all candidates, so polymorphic instantiation is handled semantically rather than by collapsing all type variables to one simple type. Erasure removes only type syntax and annotations. Every full reduction step either produces an erased untyped-β step or leaves the erasure unchanged while decreasing the number of explicit type abstractions and applications. A lexicographic reflection argument therefore yields strong normalization for the original System F term.

The proof proceeds in three stages. First, reducibility candidates establish strong normalization of the untyped erasure of every well-typed term. Second, each System F reduction is shown either to induce an untyped β-step after erasure or to preserve the erasure while strictly decreasing the amount of explicit type syntax. Finally, a lexicographic argument reflects strong normalization of the erasure back to the Church-style calculus.

Earlier formalizations include Altenkirch's LEGO development [1], the ATS/LF development of Donnelly and Xi [4], and the Isabelle/HOL development of Popescu, Gunter, and Osborn [6]. The latter formalization uses Curry-style terms and HOAS on top of FOAS. The present entry instead treats Church-style syntax directly: term and type binders use de Bruijn indices, type-β is an explicit reduction rule, and the erasure proof reflects normalization with an explicit type-syntax measure. This distinguishes the calculus, binding representation, and proof architecture without claiming novelty for the classical normalization theorem.

We also considered basing the development on Berghofer's `StrongNorm` formalization in the Isabelle/HOL `HOL-Proofs-Lambda` library [2]. That development provides a closely related de Bruijn treatment of strong normalization for the simply typed lambda calculus, including lifting, substitution, subject reduction, and an inductive characterization of terminating terms. It has only term abstraction/application, simple arrow types, and term-β, however; it does not cover polymorphic type abstraction/application, impredicative universal types, or type-β reduction. The present entry therefore follows its de Bruijn and substitution discipline while developing a System-F-specific reducibility-candidate semantics and an erasure/reflection argument for explicit type reductions. Berghofer's POPLmark AFP entry [3] is similarly valuable for the de Bruijn treatment of binding and substitution, but it formalizes the type-safety infrastructure for System $F_{<:}$, not strong normalization. We use these developments as methodological references rather than claiming to extend either source by direct import.

The development also proves subject reduction independently of the normalization argument. It is included both as a useful metatheorem and as an independent validation of the substitution infrastructure; the normalization proof does not invoke preservation. Typed weakening and term/type substitution lemmas validate the de Bruijn operations at both term-β and type-β redexes, and the preservation theorem provides an independent consistency check on the operational semantics.

theory *System-F*
  imports *Main*
begin

datatype *dty* =
    TyVar nat
  | *TyArr dty dty*

```
  | TyAll dty

datatype dtm =
    TmVar nat
  | TmApp dtm dtm
  | TmLam dty dtm
  | TmTAbs dtm
  | TmTApp dtm dty

fun ty-shift :: nat ⇒ dty ⇒ dty where
  ty-shift k (TyVar n) =
    (if n < k then TyVar n else TyVar (Suc n))
| ty-shift k (TyArr A B) = TyArr (ty-shift k A) (ty-shift k B)
| ty-shift k (TyAll A) = TyAll (ty-shift (Suc k) A)

fun ty-subst :: nat ⇒ dty ⇒ dty ⇒ dty where
  ty-subst k S (TyVar n) =
    (if n < k then TyVar n else
      if n = k then S else TyVar (n − 1))
| ty-subst k S (TyArr A B) =
    TyArr (ty-subst k S A) (ty-subst k S B)
| ty-subst k S (TyAll A) =
    TyAll (ty-subst (Suc k) (ty-shift 0 S) A)

definition ty-subst0 :: dty ⇒ dty ⇒ dty
where
  ty-subst0 S A = ty-subst 0 S A

fun tm-shift :: nat ⇒ dtm ⇒ dtm where
  tm-shift k (TmVar n) =
    (if n < k then TmVar n else TmVar (Suc n))
| tm-shift k (TmApp M N) = TmApp (tm-shift k M) (tm-shift k N)
| tm-shift k (TmLam A M) = TmLam A (tm-shift (Suc k) M)
| tm-shift k (TmTAbs M) = TmTAbs (tm-shift k M)
| tm-shift k (TmTApp M A) = TmTApp (tm-shift k M) A

fun tm-ty-shift :: nat ⇒ dtm ⇒ dtm where
  tm-ty-shift k (TmVar n) = TmVar n
| tm-ty-shift k (TmApp M P) =
    TmApp (tm-ty-shift k M) (tm-ty-shift k P)
| tm-ty-shift k (TmLam A M) =
    TmLam (ty-shift k A) (tm-ty-shift k M)
| tm-ty-shift k (TmTAbs M) =
    TmTAbs (tm-ty-shift (Suc k) M)
| tm-ty-shift k (TmTApp M A) =
    TmTApp (tm-ty-shift k M) (ty-shift k A)

fun tm-subst :: nat ⇒ dtm ⇒ dtm ⇒ dtm where
  tm-subst k N (TmVar n) =
```

```
    (if n < k then TmVar n else
      if n = k then N else TmVar (n − 1))
| tm-subst k N (TmApp M P) =
    TmApp (tm-subst k N M) (tm-subst k N P)
| tm-subst k N (TmLam A M) =
    TmLam A (tm-subst (Suc k) (tm-shift 0 N) M)
| tm-subst k N (TmTAbs M) =
    TmTAbs (tm-subst k (tm-ty-shift 0 N) M)
| tm-subst k N (TmTApp M A) =
    TmTApp (tm-subst k N M) A

definition tm-subst0 :: dtm ⇒ dtm ⇒ dtm
where
  tm-subst0 N M = tm-subst 0 N M

fun tm-ty-subst :: nat ⇒ dty ⇒ dtm ⇒ dtm where
  tm-ty-subst k S (TmVar n) = TmVar n
| tm-ty-subst k S (TmApp M N) =
    TmApp (tm-ty-subst k S M) (tm-ty-subst k S N)
| tm-ty-subst k S (TmLam A M) =
    TmLam (ty-subst k S A) (tm-ty-subst k S M)
| tm-ty-subst k S (TmTAbs M) =
    TmTAbs (tm-ty-subst (Suc k) (ty-shift 0 S) M)
| tm-ty-subst k S (TmTApp M A) =
    TmTApp (tm-ty-subst k S M) (ty-subst k S A)

definition tm-ty-subst0 :: dty ⇒ dtm ⇒ dtm
where
  tm-ty-subst0 S M = tm-ty-subst 0 S M

definition ty-env-cons :: 'a ⇒ (nat ⇒ 'a) ⇒ nat ⇒ 'a
where
  ty-env-cons C ϱ n = (case n of 0 ⇒ C | Suc k ⇒ ϱ k)

definition tm-env-cons :: dtm ⇒ (nat ⇒ dtm) ⇒ nat ⇒ dtm
where
  tm-env-cons N δ n = (case n of 0 ⇒ N | Suc k ⇒ δ k)

definition tm-env-shift :: (nat ⇒ dtm) ⇒ nat ⇒ dtm
where
  tm-env-shift δ n =
    (case n of 0 ⇒ TmVar 0 | Suc k ⇒ tm-shift 0 (δ k))

fun tm-subst-env :: (nat ⇒ dtm) ⇒ dtm ⇒ dtm where
  tm-subst-env δ (TmVar k) = δ k
| tm-subst-env δ (TmApp M N) =
    TmApp (tm-subst-env δ M) (tm-subst-env δ N)
| tm-subst-env δ (TmLam A M) =
    TmLam A (tm-subst-env (tm-env-shift δ) M)
```

| *tm-subst-env* δ (*TmTAbs* M) = *TmTAbs* (*tm-subst-env* δ M)
| *tm-subst-env* δ (*TmTApp* M A) =
  TmTApp (*tm-subst-env* δ M) A

inductive *wf-ty* :: *nat* ⇒ *dty* ⇒ *bool* **where**
  wf-var: $n < k$ ⟹ *wf-ty* k (*TyVar* n)
| *wf-arr*: ⟦*wf-ty* k A; *wf-ty* k B⟧ ⟹
  wf-ty k (*TyArr* A B)
| *wf-all*: *wf-ty* (*Suc* k) A ⟹ *wf-ty* k (*TyAll* A)

inductive *ctx-mem* :: *dty list* ⇒ *nat* ⇒ *dty* ⇒ *bool* **where**
  ctx-zero: *ctx-mem* (A # Γ) *0* A
| *ctx-suc*: *ctx-mem* Γ k A ⟹ *ctx-mem* (B # Γ) (*Suc* k) A

inductive *typing* :: *nat* ⇒ *dty list* ⇒ *dtm* ⇒ *dty* ⇒ *bool*
  (‹- ; - ⊢ - : -› [*60*,*60*,*60*,*60*] *60*) **where**
  ty-var: *ctx-mem* Γ n A ⟹ *typing* k Γ (*TmVar* n) A
| *ty-app*: ⟦*typing* k Γ M (*TyArr* A B); *typing* k Γ N A⟧
  ⟹ *typing* k Γ (*TmApp* M N) B
| *ty-lam*: ⟦*wf-ty* k A; *typing* k (A # Γ) M B⟧
  ⟹ *typing* k Γ (*TmLam* A M) (*TyArr* A B)
| *ty-tabs*: *typing* (*Suc* k) (*map* (*ty-shift* *0*) Γ) M B
  ⟹ *typing* k Γ (*TmTAbs* M) (*TyAll* B)
| *ty-tapp*: ⟦*typing* k Γ M (*TyAll* B); *wf-ty* k A⟧
  ⟹ *typing* k Γ (*TmTApp* M A) (*ty-subst0* A B)

inductive *beta* :: *dtm* ⇒ *dtm* ⇒ *bool*
  (‹- ⟶$_\beta$ -› [*80*,*80*] *80*) **where**
  beta-appL: M ⟶$_\beta$ M' ⟹
  TmApp M N ⟶$_\beta$ *TmApp* M' N
| *beta-appR*: N ⟶$_\beta$ N' ⟹
  TmApp M N ⟶$_\beta$ *TmApp* M N'
| *beta-lam*: M ⟶$_\beta$ M' ⟹
  TmLam A M ⟶$_\beta$ *TmLam* A M'
| *beta-tabs*: M ⟶$_\beta$ M' ⟹
  TmTAbs M ⟶$_\beta$ *TmTAbs* M'
| *beta-tapp*: M ⟶$_\beta$ M' ⟹
  TmTApp M A ⟶$_\beta$ *TmTApp* M' A
| *beta-term*: *TmApp* (*TmLam* A M) N ⟶$_\beta$ *tm-subst0* N M
| *beta-type*: *TmTApp* (*TmTAbs* M) A ⟶$_\beta$ *tm-ty-subst0* A M

inductive *SN* :: *dtm* ⇒ *bool* **where**
  SN-intro: (⋀M'. M ⟶$_\beta$ M' ⟹ *SN* M') ⟹ *SN* M

end
theory *System-F-Subject-Reduction*
  imports *System-F*
begin

This theory records the independent type-preservation check for the de

Bruijn presentation. It is not needed by the normalization argument, but it validates the substitution operations used by both beta rules.

fun *ctx-insert* :: *nat* ⇒ *dty* ⇒ *dty list* ⇒ *dty list* **where**
  ctx-insert i B [] = [*B*]
| *ctx-insert 0 B* (*A* # Γ) = *B* # *A* # Γ
| *ctx-insert* (*Suc i*) *B* (*A* # Γ) = *A* # *ctx-insert i B* Γ

lemma *ctx-insert-Suc-cons*:
  ctx-insert (*Suc i*) *B* (*A* # Γ) = *A* # *ctx-insert i B* Γ
  ⟨*proof*⟩

lemma *map-ctx-insert*:
  map f (*ctx-insert i B* Γ) = *ctx-insert i* (*f B*) (*map f* Γ)
⟨*proof*⟩

lemma *ctx-mem-ctx-insert*:
  assumes *ctx-mem* Γ *n A*
  shows *ctx-mem* (*ctx-insert i B* Γ)
    (*if n* < *i then n else Suc n*) *A*
⟨*proof*⟩

lemma *ctx-mem-sucD*:
  ctx-mem (*B* # Γ) (*Suc n*) *A* ⟹ *ctx-mem* Γ *n A*
  ⟨*proof*⟩

lemma *ctx-mem-insert-cases*:
  assumes *ctx-mem* (*ctx-insert i C* Γ) *n A i* ≤ *length* Γ
  shows (*n* < *i* ∧ *ctx-mem* Γ *n A*) ∨
    (*n* = *i* ∧ *A* = *C*) ∨
    (*i* < *n* ∧ *ctx-mem* Γ (*n* − *1*) *A*)
⟨*proof*⟩

lemma *typing-weakening*:
  typing k Γ *M A* ⟹
    typing k (*ctx-insert i B* Γ) (*tm-shift i M*) *A*
⟨*proof*⟩

lemma *wf-ty-shift-at*:
  assumes *wf-ty k A l* ≤ *k*
  shows *wf-ty* (*Suc k*) (*ty-shift l A*)
  ⟨*proof*⟩

lemma *wf-ty-shift*:
  assumes *wf-ty k A*
  shows *wf-ty* (*Suc k*) (*ty-shift 0 A*)
  ⟨*proof*⟩

lemma *ctx-mem-map*:
  assumes *ctx-mem* Γ *n A*

shows *ctx-mem* (*map f* Γ) *n* (*f A*)
⟨*proof*⟩

lemma *ty-shift-commute*:
assumes $i \leq j$
shows *ty-shift i* (*ty-shift j A*) =
ty-shift (*Suc j*) (*ty-shift i A*)
⟨*proof*⟩

lemma *ty-subst-shift*:
assumes $l \leq k$
shows *ty-subst* (*Suc k*) (*ty-shift l S*) (*ty-shift l A*) =
ty-shift l (*ty-subst k S A*)
⟨*proof*⟩

lemma *ty-subst-ty-shift*:
ty-subst (*Suc k*) (*ty-shift 0 S*) (*ty-shift 0 A*) =
ty-shift 0 (*ty-subst k S A*)
⟨*proof*⟩

lemma *ty-subst-shift-suc*:
assumes $k \leq l$
shows *ty-subst k* (*ty-shift l S*) (*ty-shift* (*Suc l*) *A*) =
ty-shift l (*ty-subst k S A*)
⟨*proof*⟩

lemma *typing-ty-shift-at*:
assumes *typing k* Γ *M A l* ≤ *k*
shows *typing* (*Suc k*) (*map* (*ty-shift l*) Γ)
(*tm-ty-shift l M*) (*ty-shift l A*)
⟨*proof*⟩

lemma *typing-ty-shift*:
typing k Γ *M A* ⟹
typing (*Suc k*) (*map* (*ty-shift 0*) Γ)
(*tm-ty-shift 0 M*) (*ty-shift 0 A*)
⟨*proof*⟩

lemma *wf-ty-subst-pred*:
assumes *wf-ty n A wf-ty* (*n* − *1*) *S j* < *n*
shows *wf-ty* (*n* − *1*) (*ty-subst j S A*)
⟨*proof*⟩

lemma *wf-ty-subst-at*:
assumes *wf-ty* (*Suc k*) *A wf-ty k S j* ≤ *k*
shows *wf-ty k* (*ty-subst j S A*)
⟨*proof*⟩

lemma *ty-subst-shift-id*:

ty-subst k T (ty-shift k A) = A
⟨*proof*⟩

lemma *ty-subst-comp-at*:
i ≤ j ⟶
ty-subst i (ty-subst j S T)
(ty-subst (Suc j) (ty-shift i S) A) =
ty-subst j S (ty-subst i T A)
⟨*proof*⟩

lemma *ty-subst-comp*:
ty-subst j S (ty-subst 0 T A) =
ty-subst 0 (ty-subst j S T)
(ty-subst (Suc j) (ty-shift 0 S) A)
⟨*proof*⟩

lemma *typing-ty-subst-at-aux*:
assumes *typing n Γ M A n = Suc k wf-ty k S j ≤ k*
shows *typing k (map (ty-subst j S) Γ)*
(tm-ty-subst j S M) (ty-subst j S A)
⟨*proof*⟩

lemma *typing-ty-subst-at*:
assumes *typing (Suc k) Γ M A wf-ty k S j ≤ k*
shows *typing k (map (ty-subst j S) Γ)*
(tm-ty-subst j S M) (ty-subst j S A)
⟨*proof*⟩

lemma *typing-ty-subst*:
assumes *typing (Suc k) Γ M A wf-ty k S*
shows *typing k (map (ty-subst 0 S) Γ)*
(tm-ty-subst 0 S M) (ty-subst 0 S A)
⟨*proof*⟩

lemma *typing-tm-subst-at*:
assumes *typing k Γ N C*
typing k (ctx-insert i C Γ) M A
i ≤ length Γ
shows *typing k Γ (tm-subst i N M) A*
⟨*proof*⟩

lemma *typing-tm-subst0*:
assumes *typing k Γ N C*
typing k (C # Γ) M A
shows *typing k Γ (tm-subst0 N M) A*
⟨*proof*⟩

theorem *subject-reduction*:
assumes *typing k Γ M A M* $\longrightarrow_\beta$ *M′*

shows *typing k Γ M′ A*
⟨*proof*⟩

end
theory *System-F-Untyped*
imports *System-F*
begin

datatype *ulam* =
UVar nat
| *UApp ulam ulam*
| *ULam ulam*

fun *u-shift* :: *nat* ⇒ *ulam* ⇒ *ulam* **where**
u-shift k (*UVar n*) =
(*if n* < *k then UVar n else UVar* (*Suc n*))
| *u-shift k* (*UApp M N*) = *UApp* (*u-shift k M*) (*u-shift k N*)
| *u-shift k* (*ULam M*) = *ULam* (*u-shift* (*Suc k*) *M*)

fun *u-subst* :: *nat* ⇒ *ulam* ⇒ *ulam* ⇒ *ulam* **where**
u-subst k N (*UVar n*) =
(*if n* < *k then UVar n else*
if n = *k then N else UVar* (*n* − *1*))
| *u-subst k N* (*UApp M P*) =
UApp (*u-subst k N M*) (*u-subst k N P*)
| *u-subst k N* (*ULam M*) =
ULam (*u-subst* (*Suc k*) (*u-shift 0 N*) *M*)

definition *u-subst0* :: *ulam* ⇒ *ulam* ⇒ *ulam*
where *u-subst0 N M* = *u-subst 0 N M*

lemma *u-subst-shift*:
u-subst k N (*u-shift k M*) = *M*
⟨*proof*⟩

lemma *u-shift-shift*:
l ≤ *k* ⟶
u-shift l (*u-shift k M*) =
u-shift (*Suc k*) (*u-shift l M*)
⟨*proof*⟩

lemma *u-shift-subst-comm*:
l ≤ *k* ⟶
u-shift l (*u-subst k N M*) =
u-subst (*Suc k*) (*u-shift l N*) (*u-shift l M*)
⟨*proof*⟩

lemma *u-subst-comp-at*:
l ≤ *k* ⟶

u-subst l (u-subst k N P)
(u-subst (Suc k) (u-shift l N) M) =
u-subst k N (u-subst l P M)
⟨*proof*⟩

inductive *ubeta* :: *ulam* ⇒ *ulam* ⇒ *bool*
(‹- ⟶$_u$ -› [*80*,*80*] *80*) **where**
uappL: *M* ⟶$_u$ *M′* ⟹
UApp M N ⟶$_u$ *UApp M′ N*
| *uappR*: *N* ⟶$_u$ *N′* ⟹
UApp M N ⟶$_u$ *UApp M N′*
| *ulam*: *M* ⟶$_u$ *M′* ⟹
ULam M ⟶$_u$ *ULam M′*
| *ured*: *UApp* (*ULam M*) *N* ⟶$_u$ *u-subst0 N M*

inductive *uSN* :: *ulam* ⇒ *bool* **where**
uSN-intro: (⋀*M′*. *M* ⟶$_u$ *M′* ⟹ *uSN M′*) ⟹ *uSN M*

lemma *uSN-preserved*:
assumes *uSN M* **and** *M* ⟶$_u$ *M′*
shows *uSN M′*
⟨*proof*⟩

lemma *u-subst-beta-at*:
assumes *M* ⟶$_u$ *M′*
shows *u-subst k N M* ⟶$_u$ *u-subst k N M′*
⟨*proof*⟩

lemma *uSN-subst-beta*:
assumes *M* ⟶$_u$ *M′*
shows *u-subst0 N M* ⟶$_u$ *u-subst0 N M′*
⟨*proof*⟩

lemma *uSN-reflect-subst*:
assumes *uSN* (*u-subst0* (*UVar 0*) *M*)
shows *uSN M*
⟨*proof*⟩

inductive *uFst* :: *ulam* ⇒ *ulam* ⇒ *bool* **where**
uFst-intro: *uFst* (*UApp M N*) *M*

lemma *uSN-of-fst*:
assumes *uSN* (*UApp M N*)
shows *uSN M*
⟨*proof*⟩

lemma *udouble-SN-aux*:
assumes *a*: *uSN a*
and *b*: *uSN b*

and *hyp*: $\bigwedge x\ z.$
$[\![\bigwedge y.\ x \longrightarrow_u y \Longrightarrow uSN\ y;$
$\bigwedge y.\ x \longrightarrow_u y \Longrightarrow P\ y\ z;$
$\bigwedge u.\ z \longrightarrow_u u \Longrightarrow uSN\ u;$
$\bigwedge u.\ z \longrightarrow_u u \Longrightarrow P\ x\ u]\!] \Longrightarrow P\ x\ z$
shows $P\ a\ b$
⟨*proof*⟩

lemma *udouble-SN*[*consumes 2*]:
assumes *a*: $uSN\ a$
and *b*: $uSN\ b$
and *c*: $\bigwedge x\ z.$
$[\![\bigwedge y.\ x \longrightarrow_u y \Longrightarrow P\ y\ z;$
$\bigwedge u.\ z \longrightarrow_u u \Longrightarrow P\ x\ u]\!] \Longrightarrow P\ x\ z$
shows $P\ a\ b$
⟨*proof*⟩

definition *uneutral* :: *ulam* ⇒ *bool* **where**
$uneutral\ M \longleftrightarrow$
$(\exists k.\ M = UVar\ k) \vee$
$(\exists P\ Q.\ M = UApp\ P\ Q)$

The neutral terms used by the candidate condition are variables and arbitrary applications. In particular, applications are considered neutral even when their head is an abstraction. This deliberately broad formulation makes the candidate closure stable under the full compatible reduction relation; the redex case is handled explicitly when the saturation lemmas analyze the reducts of an application.

lemma *uneutral-var*:
$uneutral\ (UVar\ k)$
⟨*proof*⟩

lemma *uneutral-app*:
$uneutral\ (UApp\ M\ N)$
⟨*proof*⟩

definition *uCR1* :: *ulam set* ⇒ *bool* **where**
$uCR1\ C \longleftrightarrow (\forall M.\ M \in C \longrightarrow uSN\ M)$

definition *uCR2* :: *ulam set* ⇒ *bool* **where**
$uCR2\ C \longleftrightarrow$
$(\forall M\ M'.\ M \in C \longrightarrow M \longrightarrow_u M' \longrightarrow M' \in C)$

definition *uCR3* :: *ulam set* ⇒ *bool* **where**
$uCR3\ C \longleftrightarrow$
$(\forall M.\ uneutral\ M \longrightarrow$
$(\forall M'.\ M \longrightarrow_u M' \longrightarrow M' \in C) \longrightarrow M \in C)$

definition *ucandidate* :: *ulam set* ⇒ *bool* **where**

ucandidate C $\longleftrightarrow$ *uCR1 C* $\wedge$ *uCR2 C* $\wedge$ *uCR3 C*

definition *uSN-set* :: *ulam set* **where**
uSN-set = $\{M.\ uSN\ M\}$

lemma *uSN-set-candidate*:
ucandidate uSN-set
⟨*proof*⟩

lemma *uvar-mem-candidate*:
assumes *ucandidate C*
shows *UVar k* $\in$ *C*
⟨*proof*⟩

definition *uarr* :: *ulam set* $\Rightarrow$ *ulam set* $\Rightarrow$ *ulam set* **where**
uarr C D = $\{M.\ \forall N.\ N \in C \longrightarrow UApp\ M\ N \in D\}$

lemma *uarr-CR2*:
assumes *uCR2 D*
shows *uCR2* (*uarr C D*)
⟨*proof*⟩

lemma *uarr-CR1*:
assumes *uCR1 C* **and** *uCR1 D* **and** *uCR3 C*
shows *uCR1* (*uarr C D*)
⟨*proof*⟩

lemma *uarr-CR3*:
assumes *uCR1 C* **and** *uCR2 C* **and** *uCR3 C*
and *uCR3 D*
shows *uCR3* (*uarr C D*)
⟨*proof*⟩

lemma *uarr-candidate*:
assumes *ucandidate C* **and** *ucandidate D*
shows *ucandidate* (*uarr C D*)
⟨*proof*⟩

lemma *ubeta-lam-cases*:
assumes *ULam M* $\longrightarrow_u$ *P*
obtains *M′* **where** *M* $\longrightarrow_u$ *M′* **and** *P* = *ULam M′*
⟨*proof*⟩

The abstraction lemma is the semantic heart of the untyped development. Its premise says that substituting every argument from the domain candidate into the body produces an element of the codomain candidate. The proof first obtains strong normalization of the body by testing it with a fresh variable, and then uses simultaneous induction on body and argument reductions to establish the candidate condition for the application of the

abstraction.

lemma *uabs-RED*:
  assumes *C*: *ucandidate C*
    and *D*: *ucandidate D*
    and *asm*: $\forall$ *s*. *s* $\in$ *C* $\longrightarrow$ *u-subst0 s M* $\in$ *D*
  shows *ULam M* $\in$ *uarr C D*
⟨*proof*⟩

end
theory *System-F-Erasure*
  imports *System-F System-F-Untyped*
begin

fun *f-erase* :: *dtm* $\Rightarrow$ *ulam* **where**
  f-erase (*TmVar n*) = *UVar n*
| *f-erase* (*TmApp M N*) = *UApp* (*f-erase M*) (*f-erase N*)
| *f-erase* (*TmLam A M*) = *ULam* (*f-erase M*)
| *f-erase* (*TmTAbs M*) = *f-erase M*
| *f-erase* (*TmTApp M A*) = *f-erase M*

lemma *f-erase-tm-shift*:
  f-erase (*tm-shift k M*) = *u-shift k* (*f-erase M*)
  ⟨*proof*⟩

lemma *f-erase-tm-ty-shift*:
  f-erase (*tm-ty-shift k M*) = *f-erase M*
  ⟨*proof*⟩

lemma *f-erase-tm-subst*:
  f-erase (*tm-subst k N M*) =
    u-subst k (*f-erase N*) (*f-erase M*)
  ⟨*proof*⟩

lemma *f-erase-tm-ty-subst*:
  f-erase (*tm-ty-subst k S M*) = *f-erase M*
  ⟨*proof*⟩

lemma *f-erase-tm-subst0*:
  f-erase (*tm-subst0 N M*) =
    u-subst0 (*f-erase N*) (*f-erase M*)
  ⟨*proof*⟩

lemma *f-erase-tm-ty-subst0*:
  f-erase (*tm-ty-subst0 S M*) = *f-erase M*
  ⟨*proof*⟩

fun *f-type-count* :: *dtm* $\Rightarrow$ *nat* **where**
  f-type-count (*TmVar n*) = *0*
| *f-type-count* (*TmApp M N*) = *f-type-count M* + *f-type-count N*

| *f-type-count* (*TmLam A M*) = *f-type-count M*
| *f-type-count* (*TmTAbs M*) = *Suc* (*f-type-count M*)
| *f-type-count* (*TmTApp M A*) = *Suc* (*f-type-count M*)

lemma *f-type-count-tm-ty-subst*:
f-type-count (*tm-ty-subst k S M*) = *f-type-count M*
⟨*proof*⟩

lemma *f-type-count-tm-ty-subst0*:
f-type-count (*tm-ty-subst0 S M*) = *f-type-count M*
⟨*proof*⟩

lemma *f-beta-progress*:
assumes $M \longrightarrow_\beta M'$
shows *ubeta* (*f-erase M*) (*f-erase M'*) ∨
f-erase M = *f-erase M'* ∧
f-type-count M' < *f-type-count M*
⟨*proof*⟩

lemma *fSN-reflect*:
assumes *uSN* (*f-erase M*)
shows *SN M*
⟨*proof*⟩

end
theory *System-F-Normalization*
imports *System-F-Erasure*
begin

definition *f-env-ok* :: (*nat* ⇒ *ulam set*) ⇒ *bool* **where**
f-env-ok ϱ ⟷ (∀ *k*. *ucandidate* (ϱ *k*))

definition *f-all-sem* :: (*ulam set* ⇒ *ulam set*) ⇒ *ulam set* **where**
f-all-sem F = {*M*. ∀ *C*. *ucandidate C* ⟶ *M* ∈ *F C*}

lemma *f-all-candidate*:
assumes *H*: ∀ *C*. *ucandidate C* ⟶ *ucandidate* (*F C*)
shows *ucandidate* (*f-all-sem F*)
⟨*proof*⟩

definition *f-env-insert* ::
nat ⇒ *ulam set* ⇒ (*nat* ⇒ *ulam set*) ⇒
nat ⇒ *ulam set*
where
f-env-insert k C ϱ *n* =
(*if n* < *k then* ϱ *n else*
if n = *k then C else* ϱ (*n* − *1*))

lemma *f-env-insert-zero*:

f-env-insert 0 C ϱ = ty-env-cons C ϱ
⟨*proof*⟩

lemma *f-env-insert-suc-cons*:
f-env-insert (Suc k) C (ty-env-cons D ϱ) =
ty-env-cons D (f-env-insert k C ϱ)
⟨*proof*⟩

lemma *f-env-ok-cons*:
assumes *f-env-ok ϱ* **and** *ucandidate C*
shows *f-env-ok (ty-env-cons C ϱ)*
⟨*proof*⟩

fun *f-sem* :: *dty* ⇒ (*nat* ⇒ *ulam set*) ⇒ *ulam set* **where**
f-sem (TyVar k) ϱ = ϱ k
| *f-sem (TyArr A B) ϱ =*
uarr (f-sem A ϱ) (f-sem B ϱ)
| *f-sem (TyAll A) ϱ =*
f-all-sem (λC. f-sem A (ty-env-cons C ϱ))

lemma *f-sem-candidate*:
f-env-ok ϱ ⟶ ucandidate (f-sem A ϱ)
⟨*proof*⟩

lemma *f-sem-ty-shift*:
f-sem (ty-shift k A) (f-env-insert k C ϱ) =
f-sem A ϱ
⟨*proof*⟩

lemma *f-sem-ty-subst*:
f-sem (ty-subst k S A) ϱ =
f-sem A (f-env-insert k (f-sem S ϱ) ϱ)
⟨*proof*⟩

definition *f-tenv-cons* :: *ulam* ⇒ (*nat* ⇒ *ulam*) ⇒
nat ⇒ *ulam*
where
f-tenv-cons N η n =
(*case n of 0* ⇒ *N* | *Suc k* ⇒ *η k*)

definition *f-tenv-shift* :: (*nat* ⇒ *ulam*) ⇒ *nat* ⇒ *ulam*
where
f-tenv-shift η n =
(*case n of 0* ⇒ *UVar 0* | *Suc k* ⇒ *u-shift 0* (*η k*))

definition *f-env-subst* ::
nat ⇒ *ulam* ⇒ (*nat* ⇒ *ulam*) ⇒
nat ⇒ *ulam*
where

f-env-subst k N η n = u-subst k N (η n)

lemma *f-env-subst-shift*:
f-env-subst (Suc k) (u-shift 0 N) (f-tenv-shift η) =
f-tenv-shift (f-env-subst k N η)
⟨*proof*⟩

lemma *f-env-subst-shift0*:
f-env-subst 0 N (f-tenv-shift η) =
f-tenv-cons N η
⟨*proof*⟩

fun *f-interp* :: *(nat ⇒ ulam) ⇒ dtm ⇒ ulam* **where**
f-interp η (TmVar n) = η n
| *f-interp η (TmApp M N) =*
UApp (f-interp η M) (f-interp η N)
| *f-interp η (TmLam A M) =*
ULam (f-interp (f-tenv-shift η) M)
| *f-interp η (TmTAbs M) = f-interp η M*
| *f-interp η (TmTApp M A) = f-interp η M*

lemma *f-interp-subst*:
u-subst k N (f-interp η M) =
f-interp (f-env-subst k N η) M
⟨*proof*⟩

lemma *f-tenv-shift-id*:
f-tenv-shift (λn. UVar n) = (λn. UVar n)
⟨*proof*⟩

lemma *f-interp-id*:
f-interp (λn. UVar n) M = f-erase M
⟨*proof*⟩

lemma *f-interp-subst0-shift*:
u-subst0 N (f-interp (f-tenv-shift η) M) =
f-interp (f-tenv-cons N η) M
⟨*proof*⟩

definition *f-tenv-ok* ::
dty list ⇒ (nat ⇒ ulam) ⇒
(nat ⇒ ulam set) ⇒ bool
where
f-tenv-ok Γ η ϱ ⟷
(∀ n A. ctx-mem Γ n A ⟶
η n ∈ f-sem A ϱ)

lemma *f-ctx-mem-mapE*:
ctx-mem (map (ty-shift 0) Γ) n B ⟶

$(\exists A.\ \mathit{ctx\text{-}mem}\ \Gamma\ n\ A \wedge B = \mathit{ty\text{-}shift}\ 0\ A)$
⟨*proof*⟩

lemma *f-tenv-ok-cons*:
assumes *hctx*: *f-tenv-ok* $\Gamma\ \eta\ \varrho$
and *hval*: $N \in$ *f-sem* $A\ \varrho$
shows *f-tenv-ok* $(A\ \#\ \Gamma)$ $(\mathit{f\text{-}tenv\text{-}cons}\ N\ \eta)\ \varrho$
⟨*proof*⟩

lemma *f-tenv-ok-ty-shift*:
assumes *hctx*: *f-tenv-ok* $\Gamma\ \eta\ \varrho$
and *henv*: *f-env-ok* ϱ
and *hcan*: *ucandidate* C
shows *f-tenv-ok* $(\mathit{map}\ (\mathit{ty\text{-}shift}\ 0)\ \Gamma)\ \eta$
$(\mathit{ty\text{-}env\text{-}cons}\ C\ \varrho)$
⟨*proof*⟩

definition *f-prop* ::
dty list ⇒ *dtm* ⇒ *dty* ⇒ *bool*
where
f-prop $\Gamma\ M\ A \longleftrightarrow$
$(\forall \varrho\ \eta.$
f-env-ok $\varrho \longrightarrow$
f-tenv-ok $\Gamma\ \eta\ \varrho \longrightarrow$
f-interp $\eta\ M \in$ *f-sem* $A\ \varrho)$

lemma *f-fundamental*:
typing $k\ \Gamma\ M\ A \Longrightarrow$ *f-prop* $\Gamma\ M\ A$
⟨*proof*⟩

lemma *f-identity-tenv-ok*:
f-tenv-ok Γ $(\lambda n.\ \mathit{UVar}\ n)$ $(\lambda\text{-}.\ \mathit{uSN\text{-}set})$
⟨*proof*⟩

lemma *f-welltyped-fundamental*:
assumes *typing* $k\ \Gamma\ M\ A$
shows *uSN* $(\mathit{f\text{-}erase}\ M)$
⟨*proof*⟩

theorem *strong-normalization*:
assumes *typing* $k\ \Gamma\ M\ A$
shows *SN* M
⟨*proof*⟩

end

References

[1] T. Altenkirch. A formalization of the strong normalization proof for system f in lego. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 1993. DOI: https://doi.org/10.1007/BFB0037095.

[2] S. Berghofer. Strong normalization for simply-typed lambda calculus. Isabelle/HOL library theory `HOL/Proofs/Lambda/StrongNorm.thy`, 2000. https://isabelle.in.tum.de/library/HOL/HOL-Proofs-Lambda/document.pdf.

[3] S. Berghofer. Poplmark challenge via de bruijn indices. *Archive of Formal Proofs*, August 2007. https://isa-afp.org/entries/POPLmark-deBruijn.html, Formal proof development.

[4] K. Donnelly and H. Xi. A formalization of strong normalization for simply-typed lambda-calculus and system f. *Electronic Notes in Theoretical Computer Science*, 174(5):109–125, 2007. DOI: https://doi.org/10.1016/j.entcs.2007.01.021.

[5] J.-Y. Girard. Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur. *Thèse de doctorat, Université Paris VII*, 1972.

[6] A. Popescu, E. L. Gunter, and C. J. Osborn. Strong normalization for system f by hoas on top of foas. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*, pages 31–40. IEEE, 2010. DOI: https://doi.org/10.1109/LICS.2010.48.